# EXHIBIT D

# CORRECT MEMORY OPERATION OF CACHE-BASED MULTIPROCESSORS†

**Christoph Scheurich and Michel Dubois**

Computer Research Institute
Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, California 90089-0781

## ABSTRACT

This paper shows that cache coherence protocols can implement indivisible synchronization primitives reliably and can also enforce sequential consistency. Sequential consistency provides a commonly accepted model of behavior of multiprocessors. We derive a simple set of conditions needed to enforce sequential consistency in multiprocessors. These conditions are easily applied to prove the correctness of existing cache coherence protocols that rely on one or multiple broadcast buses to enforce atomicity of updates; in these protocols, all processing elements must be connected to the broadcast buses. The conditions are also used in this paper to establish new protocols which do not rely on the atomicity of updates and therefore do not require single access buses to propagate invalidations or to perform distributed WRITEs. It is also shown how such protocols can implement atomic READ&MODIFY operations for synchronization purposes.

## 1. INTRODUCTION

The performance of shared-memory multiprocessors is limited by memory access conflicts. A usual means of reducing these conflicts is to associate a private cache with each processor [1, 2, 3]. In a multiprocessor system, the use of private caches poses the familiar and complex problem of *cache coherence:* namely, the problem of how to maintain data coherence among the various copies of data which can reside in multiple caches and main memory, in a manner that is completely transparent to the user of the machine. This latter condition is highly desirable in a general-purpose environment. Many verified solutions to the problem of multiprocessor cache coherence have been proposed and several have been successfully implemented. Much effort has been devoted to the minimization of the overhead traffic necessary to maintain coherence on single bus systems [4, 5, 6]. As a result of minimizing overall traffic, the complexity of the coherence protocols has increased. To prove most of these efficient protocols correct has become exceedingly difficult.

† This research is supported by an NSF Research Initiation Grant No. DMC-8505328.



Cache coherence protocols must provide for the execution of indivisible READ&MODIFY instructions, such as the TEST&SET instruction. The implementation of indivisible (atomic) instruction sequences greatly simplifies interprocessor synchronization. Another desirable property of multiprocessors is sequential consistency. Sequential consistency is the strongest possible requirement for the logical behavior of multiprocessor systems. In this paper a condition for sequential consistency in the context of cache-based multiprocessors is derived. We also show how the condition can be applied to meet the following goals of a system designer:

- Demonstrate that a cache coherence protocol is correct by showing that it is sequentially consistent.
- Design new protocols for cache-based multiprocessors, which *do not* require a single-access broadcast interconnection to propagate data updates and invalidations.
- Implement the correct execution of indivisible instruction sequences, such as the TEST&SET instruction, in any system that uses an *ownership* protocol.

In traditional cache coherence schemes there is a single-access broadcast system, usually a bus. Traffic through the bus is a direct result of the communication and sharing of data between processors. It is unlikely that this traffic can be reduced indefinitely through more "clever" coherence protocols, mostly because in a general-purpose environment, tasks are diverse and the sharing of data between them can take different forms; more importantly, there are intrinsic lower bounds on the amount of sharing required by algorithms to solve certain problems.

In this paper we avoid the notion of *memory coherence* to analyze cache-based multiprocessors, but rather use the concept of *sequential consistency*. The reason for our approach lies in the fact that the traditional definition of memory coherence fails to provide sufficient indication on how to enforce it in systems where WRITEs are not atomic. In systems where WRITEs are not atomic, updates may become *available* at different times to different processors. To accomplish atomic updates, either single buses are used to perform all global accesses or a dedicated invalidation bus is employed to broadcast invalidations only. The requirement for at least one bus to which *all* processors must be connected inherently limits the possible number of processors in the system. This limitation is a result of the electrical problems posed by connecting many devices to a single bus, as well as of the increase of contention on the bus when the bus becomes saturated with requests.

In a previous paper [7], we have described how generic multiprocessor systems can operate correctly depending on whether the user expects the system to behave in a *weakly ordered* or in a *strongly ordered* fashion. In a *weakly ordered* system, coherence is only enforced at synchronization points. Between two successive synchronization points, a programmer does not make any assumption about the order in which updates are observed by processors.* Bitar and Despain [8] have shown how weak ordering can be easily upheld by implementing synchronization primitives (called *hard atoms*) correctly in cache-based systems with a global broadcast capability.

A system in which memory accesses are strongly ordered is also sequentially consistent. Sequential consistency is the strongest requirement possible on the ordering of events in a multiprocessor because it imposes the condition that processors may not observe STOREs in different orders. Systems that behave in a strongly ordered manner are very conventional. They include the C.mmp [9], the Synapse System N+1 [10], and the proposed Spur system [11]. All parallel programming languages we know of present the programmer with a model of execution which either adheres to sequential consistency [12] or a model of execution which is not violated if the system is sequentially consistent. While synchronized algorithms rely on explicit synchronization for correctness, there exists also a class of algorithms, called *asynchronous* in [13] and [14], in which all synchronizations have been removed and which work because processes can read and update shared variables without synchronizing.

Several papers in the literature have applied the sequential consistency condition to show the correctness of multiprocessor behavior. Lamport has applied similar reasoning to distributed, message-based systems [15] and memory accesses in shared memory systems with no cache [16]. Rudolph and Segall proved the correctness of a *snoopy cache* protocol on a single bus. Collier [17] has developed a theoretical framework based on possible execution graphs to study the coherence properties and ordering of events in a multiprocessor where each processor owns a replica of the whole shared memory. Approaches similar to Collier's have been used to analyze the correct sharing of registers by asynchronous hardware [18].

The conditions for correctness derived in [7] were improved upon in [19]. Using these conditions, we derive, in this paper, specific rules under which cache-based multiprocessors remain sequentially consistent. By applying these rules, existing and proposed "cache coherence" protocols can be demonstrated correct (in the context of sequential consistency). Furthermore, new protocols can be developed which need not rely on the atomicity of updates and therefore do not require broadcast systems. Such systems are likely to be much more scalable than conventional bus-based systems.

In order that the paper be self-contained, we recall, in section two, some basic definitions that were presented in [7] for generic multiprocessors. In section three, new definitions are introduced for the specific case of cache-based systems. A simple new condition for sequential consistency is presented; this condition is powerful because it can easily be applied to demonstrate that non-trivial protocols are sequentially consistent. In section four, the derived conditions for sequential consistency are applied to two proposed cache coherence schemes: one with a single system bus and one with a coherence table and one single invalidation bus. In section five a system is described that does not rely on atomic updates to be sequentially consistent or to execute indivisible synchronization primitives. We conclude in section six.

* The only condition on ordering of memory accesses is that dependencies within each individual instruction stream must be enforced.

## 2. SEQUENTIAL CONSISTENCY AND COHERENCE

Most concepts elaborated upon in this section were introduced and discussed in a previous paper. A general assumption from the programmer or the compiler of a multiprocessor is that the machine is sequentially consistent (see, for example, [12] for a discussion of sequential consistency in high-level concurrent languages).

**Definition 2.1: Sequential Consistency**

Lamport defines sequential consistency as follows.

> "[A system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by its program."

Sequential consistency defines a system in which operations are performed in individual program order but which does not make any further assumptions about the order in which operations are executed. With the exception of some special-purpose machines, most multiprocessor systems adhere to the sequential consistency model. More importantly, sequential consistency is the model of execution which most users assume, since parallel languages directly imply it.

The common model used to define correct execution of cache-based multiprocessors is, however, not sequential consistency but *memory coherence*. Censier and Feautrier describe a coherent memory system as follows.

> "A memory scheme is coherent if the value returned on a LOAD instruction is always the value given by the latest STORE instruction with the same address."

In the context of a complex multiprocessor, the notion of "latest STORE" is vague. As will be shown in the remainder of this section, a memory access may be in several different stages of partial completion before it is *entirely* executed. Depending on how one defines the execution of a memory access, the above definition is equivalent or different from the definition of sequential consistency.

When examined individually, processors and memories of multiprocessor systems can easily be proven to function correctly. But when several processors and memories are active concurrently, the additional factor of sequencing of events must be taken into consideration. In multiprocessor systems we are primarily concerned with *events* that can affect the overall system. If we disregard interprocessor interrupts, then shared memory accesses are such events which can affect the entire system. STOREs change the state of a memory word, which when subsequently read by another processor can affect that processor's future activity. LOADs enable processors to read shared memory words and thus enable them to be affected by other processors. STORE and LOAD events, however, can not always be viewed to occur at specific instances, but rather

sometimes occur relative to each other, independent of absolute real time.

**Definition 2.2: Memory Request Initiating, Issuing, and Performing**

> A memory access request is *initiated* when a processor has sent the request and the completion of the request is out of its control. An initiated request is *issued* when it has left the processor environment* and is in transit in the memory system. A LOAD is considered *performed* at a point in time when the issuing of a STORE to the same address cannot affect the value returned by the LOAD. A STORE on X by processor i is considered *performed* at a point in time when a subsequently issued LOAD to the same address returns the value defined by a STORE in the sequence {Si(X)}+.

In this definition, we denote LOAD and STORE accesses by processor i on variable X as Li(X) and Si(X), respectively. In a system where memory accesses are atomic, the STORE events on one variable can be ordered based on the physical time at which they are performed. The notation {Si(X)}+ is used to denote the sequence of STOREs on X following the STORE Si(X) and including Si(X) in the ordering based on the time when the STOREs are performed. Similarly, the notation {Si(X)}- denotes the sequence of STOREs on X preceding Si(X) but not including Si(X).

Definition 2.2 is relevant whether or not memory accesses are atomic. Memory accesses are atomic in a system if the value stored by a WRITE operation becomes readable at the same time for all processors. When memory accesses are not atomic, we define the notion of *performed with respect to a processor*.

**Definition 2.3: Performing with respect to a processor**

> A STORE by processor i is considered *performed with respect to processor k* at a point in time when a subsequently issued LOAD to the same address by processor k returns the value defined by a STORE in the sequence {Si(X)/k}+.
> Similarly, a LOAD by processor i is considered performed with respect to processor k, at a point in time when a subsequently issued STORE to the same address by processor k cannot affect the value returned by the LOAD.

The definitions of {Si(X)/k}+ and of {Si(X)/k}- are similar to the definitions of {Si(X)}+ and {Si(X)}-. However, the STOREs on X are ordered according to the time when they are performed with respect to processor k.

From the above definition it should now be clear why we consider some events not to occur at instances in time, but rather relative to each other. In some architectures, for example, at time $t$ a STORE may have been performed with respect to processor $x$ but not yet with respect to processor $y$. It is certainly futile to attempt to associate a particular point in time with the STORE event.

* The processor environment includes the CPU and local buffers.

**Definition 2.4: Performing an access globally**

> A STORE is *globally performed* when it is performed with respect to all processors. A LOAD is *globally performed* if it is performed with respect to all processors and if the STORE which is the source of the returned value has been globally performed.

Once a STORE is globally performed, no issued LOAD by any processor can return an old value which was valid before the STORE took place. There is a subtle difference between a LOAD *performed with respect to all processors* and a *globally performed* LOAD. Once a LOAD is performed with respect to all processors, the value it returns is fixed and cannot be altered, independent of any action of any processor. When a LOAD is globally performed, it is additionally true that any other LOAD issued subsequently by any processor cannot return a word which is "older" (i.e., a word in {Si(X)}-) than the word returned by the performed LOAD.

A sufficient condition for sequential consistency is derived in this section. This is a new condition, which was not given in [7].

**Condition 2.1: Sequential Consistency**

> Sequential consistency is satisfied in *any* system if an access may not be performed with respect to any processor until the previous access by the same processor has been globally performed and if accesses of each individual processor are globally performed in program order.

This condition is powerful because it is easily applicable in practical systems and provides a solid basis for proving non-trivial cache coherence protocols correct. Note that this condition is sufficient but not necessary.

From condition 2.1, we can say that any system is sequentially consistent if the following are true:

**(A) All processors issue memory accesses in program order.**

**(B) If a processor issues a STORE, then the processor may not issue another memory access until the value written has become *accessible* by *all other processors*.**

**(C) If a processor issues a LOAD, then the processor may not issue another memory access until the value which is to be read has both been *bound* to the LOAD operation and become accessible to all other processors.**

The following is an informal proof of the above conditions:

> Multiprocessors such as the C.mmp are classified as sequentially consistent [16] because of the following reasons:
>
> 1) Processors execute statements in program order.
>
> 2) Memory accesses are performed one-by-one; an access i is not issued until access i-1 has been completed.
>
> 3) STOREs are atomic: i.e., they become readable at the same time by all processors.

We note that no notion of timing is present in the above conditions; that is, even if accesses are delayed by a random but finite amount of time, the system remains sequentially consistent. Point (3) is not a strong condition, if the system *somehow* maintains conventional memory coherence, such as provided by an invalidation bus scheme. Hence, a system governed by condition 2.1 behaves as dictated by the above three points for the following reasons:

(I) Point (1) is maintained per definition of condition 2.1.

(II) Point (2) is maintained per definition as well, since an access cannot be *more complete* than when it has been performed.

(III) When condition 2.1 is upheld, the effect of READs on a variable is delayed until the STORE defining the value is performed; therefore, for all practical purposes, one can consider that the STOREs are performed atomically at the point in time when the value is readable by all processors.

The outcome of (III) depends on whether or not a WRITE to the memory location to be READ is presently in progress. A READ x may not proceed until the latest WRITE x ← D has *settled*. This means that at any time a processor may read a word x, reflecting the $n^{th}$ update of x, but no processor at that time is allowed to read x, reflecting the $(n+1)^{th}$ update, if this update is still in progress.

## 3. READ PATHS, OWNERS AND KEEPERS IN COHERENCE PROTOCOLS

In the previous section, the different states in which a memory access may be at any time were defined. In a system that maintains single copies of data, it is simple to analyze in what state an access is at any time. However, in systems where multiple copies of data exist, the different states in which an access may be are not easily analyzed. For such systems, READs and WRITEs often imply other activities such as invalidations or broadcasts of data. By redefining what exactly any READ or WRITE implies in cache-based systems and by classifying copies of data, it becomes simple to recognize when exactly any access is *performed* or *globally performed*.

### *3.1 Read paths*

In a conventional cache-based system with invalidations [6], a WRITE is performed with respect to a processor k when both the WRITE has been queued at the main memory and the cache which belongs to processor k has been invalidated. If one of the two conditions has not been met, then stale data may be read by a processor. Hence, *performing a WRITE with respect to a processor k* implies not only that a copy of the data must be updated, but also that processor k must be made aware of where to find the data. We refer to the pointer to valid data as a *read path* that leads from the cache to main memory. A WRITE consists of setting read paths and modifying the data. In the normal invalidation approach, a read path to main memory is *implied by invalidating the block*. In other schemes, the read path may implicitly point to a table that in turn explicitly points to a valid copy of data contained in some other cache.

A WRITE is globally performed once the read paths of all caches have been modified (if necessary) to point to valid copies of data and once all copies of data have been updated to reflect the WRITE operation. A READ is globally performed when it is guaranteed to return a particular value and when the WRITE which *caused* that value has been globally performed.

### *3.2 Owners and keepers*

When a cache or main memory contains a valid block, then this block may be the only copy or one of several valid copies of the block. If several copies of the block exist, then the storage devices containing them are *keepers* of the block. If the cache or memory contains the only copy of the block, then the storage device is the *owner* of the block. Any owner is also a keeper. If a WRITE is performed to a keeper of a block, then all other keepers must be updated. If a WRITE is performed to an owner of a block, then no further action need be taken to maintain sequential consistency. Data may be read from any keeper or owner. If a READ operation obtains the copy of the block from an owner and the block is allocated, the READ must be handled specially since the owner will cease being an owner. If there is an owner, then there is always only a single owner. In a write-through system, main memory is always a keeper, and, if there is one at all, main memory is always the only possible owner. Hence, we distinguish among four different events which occur in cache-based multiprocessor systems, regardless of the cache coherence scheme used.

(1) MODIFICATION OF STORAGE. This event occurs at either a cache or main memory. The storage device must either be a keeper or an owner, or will become one of the two. A single WRITE, as perceived by the processor, may cause several such memory modifications, such as a broadcast of WRITEs if several existing keepers need to be updated to maintain coherence. Often, a WRITE consists not only of the modification of a word but also of the transfer of an entire block which consists of several words. This may be the case, for example, in a write-back system when a dirty block is replaced in a cache.

(2) MODIFICATION OF A READ PATH. This modification makes known to the processor the location of the copy which is currently valid. Theoretically, the read path may consist of a finite number of "pointers" (i.e., some type of address to identify memories, caches, and tables) which the reading process can follow to find valid copies of a block. If a block is present and valid in a cache, then the read path is simply "nil". Usually a read path implies either: a local copy of a block (path is nil); an invalidation which implies a table lookup (path to table) which in turn either points at a valid block in another cache (path to specific cache) or main memory (path to memory); or an invalidation which directly implies that the block must be fetched from main memory (path to memory). The modification of a read path should proceed from the destination to the source, so that when a path is followed it guides the request always either to the "old" or the "new" copy of the data, but never into oblivion. A read path may only be modified by one WRITE at a time. (This serialization, however, does not mean WRITE atomicity.)

(3) FOLLOWING A READ PATH AND READING DATA. We normally do not distinguish between following a read path and actually reading. The two activities are the states in which a READ may be, and they are analogous to a READ which is issued and a READ which is performed.

(4) BECOMING AN OWNER AND RELEASING OWNERSHIP. Not all storage devices contain valid copies of all blocks at all times; that is, not all storage devices are keepers of all blocks. However, all storage devices maintain valid read paths to keepers, by default. This is the reason why a switch of keepership (i.e., becoming a keeper) is not included in our list of events. A switch of keepership manifests itself as a read path modification. For ownership changes, though, an event type is defined. Ownership changes are caused either by a processor that for some reason (usually a WRITE) wishes to acquire ownership, or because ownership may simply be lost by a cache when it replaces an owned block. Ownership by caches exists only in write-back systems, because in write-through systems main memory is always a keeper. Since an owner must be the only keeper, no cache can ever be an owner.

The above events are triggered by LOADs, STOREs, and block replacements. One STORE may cause all four events; for example, the issuing processor's cache becomes the new owner (event 4), the block is fetched from a previous keeper (event 3), the block is written to the new owner (event 1), and all other caches must be "made aware" that the block can now be found at the new owner only (event 2).

## 4. SEQUENTIAL CONSISTENCY IN BUS-BASED SYSTEMS

We wish to establish the usefulness of conditions A, B, and C defined in section two to demonstrate the correctness of cache coherence protocols.

### *4.1 Snoopy cache protocol*

In [3] Rudolph and Segall propose a cache coherence protocol of moderate complexity. In the same paper the protocol is extended to accommodate an interleaved cache system which connects each processor to several private caches, with the result that communications may be spread over several different buses. In the following, both approaches are shown to be correct.

A single bus connects all private caches of the proposed system (Fig. 1). The block size used by the caches is one word. Using a block size of a single word simplifies WRITE transactions, since any single WRITE always updates the *entire block* to a value which reflects the most recent WRITE. The system employs a write-back scheme, which provides for a write-back whenever the block is accessed by a processor which misses at the cache and therefore causes the block to be placed on the bus. A block in a cache may be in one of three states: I, R, L. State I is the invalid state; in this case, the read path points to the main memory. State R signifies that the cache is a keeper of the block, and state L means that the cache is an owner of the block.

A block which is in the invalid state has an implied read path which points to main memory. A read miss causes an access to main memory. However, if there is a cache with an L copy of the block (owner), then the main memory request of the processor that missed on the READ is interrupted by the owner. On the next bus cycle, the owner updates main memory. The interrupted processor re-issues the READ to the block. While main memory is not actually a keeper at all times, it appears that way for any cache processing a READ miss.

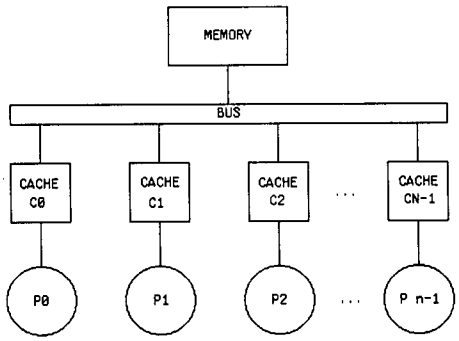



**Figure 1. A bus-based, cache-based system as proposed by Rudolph and Segall, 1984.**

In order for the protocol to satisfy conditions A, B, and C of section two (and therefore be sequentially consistent), three issues must be addressed:

1) Does each processor perform READ and WRITE accesses in program order?

2) Does a writing processor stop issuing another access before the word updated by the WRITE has become accessible by *all* processors?

3) Does a reading processor stop issuing another access before the data value that it receives by the READ has become accessible to all other processors?

Question one is not addressed in the paper [3], but it is safe to assume that it is upheld.

When a cache block is in one of the three possible states of invalid, keeper, or owner, five transactions may affect it, namely: read, write, replace, bus read, and bus write. Of these we are concerned with all transactions that either follow or modify any read path. A read path is followed in only one case: when the block is not present in the cache and a READ is executed. In this case, the read path implicitly points to main memory and is performed at main memory (even though a READ may be interrupted once to update main memory). When the READ is issued, the value returned is equally accessible by all other processors; hence, the read is performed globally.

When copies of a block are invalidated, the read path is modified under two conditions: (1) A WRITE is issued by the processor, the copy is fetched from a keeper (either an owner cache or main memory), and all read paths of all caches are modified in one bus operation to point to main memory, by means of invalidation. (2) A bus READ is issued; this causes the block to be allocated. The cache block

is now in state R (keeper) and the read path does not point to main memory any more. The recent modification of the block remains equally accessible to all processors.

When a block is in state R (keeper), read paths are modified under two conditions: (1) A WRITE is issued by the processor, the block is updated, and all other caches are invalidated such that their read paths point implicitly to main memory. This is done at the same time for all processors by means of a single bus operation. The "changed" data becomes accessible at the same time to all processors. (2) A bus WRITE is detected; this causes the keeper to invalidate its copy and therefore to set the read path to main memory. This occurs simultaneously to all keepers so that after the bus WRITE, all but the owner's read path point to main memory.

When a block is in state L (owner), read paths are modified under three conditions: (1) When the block is displaced to make space for another block (i.e., it is written back to main memory): while the write-back occurs, the bus is blocked so that no access to the same block by another processor can be interleaved; once the block is written back, it is not present in the cache anymore, so that the read path points to main memory. All other processors' caches have had read paths set to main memory all along -- no modification is necessary. (2) In case of a bus READ, the owner must supply the desired word and change its own status from owner to keeper. The owner places the block on the bus, and all other caches copy it and become keepers. Again, the block becomes accessible at the same time for all processors. (3) In case of a bus WRITE, the owner must supply the valid block and change its status from owner to invalid. The read path to the block is changed to point to main memory. All processors' caches, with the exception of the one which caused the bus WRITE, keep their read paths pointed at main memory. But, now a different owner would interrupt a bus READ or WRITE and supply the block to main memory. This change in ownership is transparent to the other caches, and they all remain equally capable of accessing the block in question.

The cache protocol has not been described in detail, but if the rules established in sections two and three are used, it should still be evident that the scheme described by Rudolph and Segall is indeed sequentially consistent. Every time a block changes state, this change is "visible" by all processors at the same time.

In the same paper, the authors propose a system with multiple buses and caches connected to an interleaved memory system (Fig. 2). Within each bus, connected cache, and memory module, the behavior of the system is equivalent to the one in Figure 1. In particular, changes of state are atomic, since every processor "watches" all buses. Therefore, according to condition 2.1, the condition for sequential consistency is simply that all accesses must be *performed* in program order. This restriction may preclude certain *apparent* advantages of the described architecture/protocol. For example, a processor $P_i$ may not issue two subsequent WRITES, $M(x) \leftarrow D1$, $M(x+1) \leftarrow D2$, concurrently, even though memory locations $M(x)$ and $M(x+1)$ always imply two different caches, buses, and memory modules. A possible incorrect outcome could be that a processor $P_j$ reads $M(x+1)$ and returns $D2$. If the system is sequentially consistent, then the program may have been written with the reasoning in mind that if $D2$ has been generated (by $P_i$), then certainly $D1$ must have been generated (also by $P_i$). But, this will not necessarily be the case, since the two WRITEs were not ordered, and the WRITE to $M(x)$ may have been delayed because of bus contention.

For the same reasoning as given above, there may not be independent FIFO access queues associated with each cache, since the sequencing of accesses between queues cannot be controlled. Hence, the interleaved cache system described by Rudolph and Segall is sequentially consistent by token of the argument made for the non-interleaved system, with the restriction that a processor's previous access must have been globally performed before the next access may be issued. (In [3] it is never proposed that two or more accesses by the same processor can be serviced concurrently by different caches.) We note that an analysis of the system based on condition 2.1 allows us to identify clearly what can and what cannot be done at each stage of the protocol.

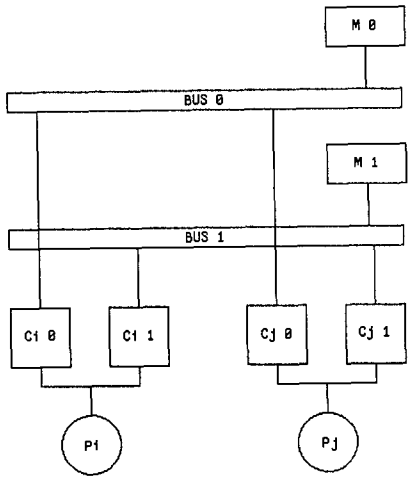



**Figure 2. A bus-based, cache-based system with interleaved caches and multiple buses [3].**

*4.2 A table-based protocol*

In [20] a multiprocessor architecture is described, that does not rely on a single system bus to propagate all accesses (Fig. 3). A global table, in the vicinity of main memory, keeps track of the states of all cached blocks. A block may be either in a shared state (RO), or in an exclusive state (RW). These states directly correspond to our keeper and owner states. If a block is in the RW state, then the table indicates who the owner is. A block without a corresponding entry in the table is owned by main memory. A bit, associated with every block frame in cache, indicates the state of the block as well.

All data transfers are propagated over an interconnection which allows several transfers to be in progress at one time. The interconnection can provide much greater bandwidth than a single-access bus as proposed in conventional architectures. A single-access, high-speed bus propagates *all* invalidations to *all* caches simultaneously. Invalidations are buffered in a FIFO buffer at each cache where they wait to be serviced.

All READ misses first interrogate the table for the state of the block. If the block is either not cached or in state RO, then the requesting cache obtains a copy of the block from main memory. If the block is in state RW, then the owner is forced to write-back the copy to main memory, and the state of the block is changed to RO. Subsequently,

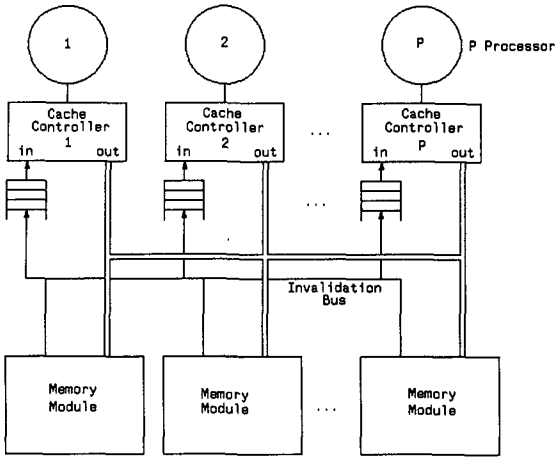



**Figure 3. A tightly-coupled cache-based system with a single invalidation bus [20].**

the cache which triggered the table interrogation becomes a keeper of the block. A WRITE proceeds immediately, if the writing processor's cache owns the block (i.e., the block is in state RW). A WRITE by processor $P_i$ causes a table interrogate if it misses at the cache or if its copy of the block is in state RO. The table interrogate can have three possible outcomes:

(1) No cached copies of the block exist. The entry is marked RW, with the owner as $P_i$'s cache. A copy of the block is transferred from main memory to $P_i$'s cache.
(2) The block is marked as RO at the table. All copies of the block are first invalidated. The block is marked as RW and a copy of it transferred from main memory to $P_i$'s cache.
(3) The block is marked as RW at the table, and $P_j$'s cache is indicated as the owner. $P_j$'s cache is forced to write back a copy of the block to main memory and is then invalidated. The table entry is modified to indicate that $P_i$'s cache is the new owner of the block. Then a copy of the block is transferred from main memory to $P_i$'s cache.

The scheme is very similar to the snoopy cache scheme, in so far as the invalidation bus guarantees serialization and atomicity of updates. However, the processor to memory interconnection provides greater communication bandwidth and is not burdened with the broadcasting of invalidations. If, for the time being, the fact that invalidates can be buffered is ignored, principles A, B, and C (from section 2) can be applied to each possible READ and WRITE situation to show that the system remains sequentially consistent. However, if we consider the possibility of buffering invalidations at each cache, the system may not remain sequentially consistent, because invalidations can modify the read paths of the caches to a block. If the invalidations are buffered and an access is considered completed when all invalidations are buffered, it is possible that a cache $i$ may process an invalidation before a cache $j$ does so. This violates both principles B and C. The WRITE which caused the invalidation is allowed to proceed after all invalidations are buffered; at this time, however, the WRITE has not been globally performed (i.e., it has not been performed with respect to processor $j$). By the same token processor $i$ may not proceed to access the invalidated block, since the WRITE which caused the latest update has not been performed with respect to processor $j$.

Buffering invalidations, though, is a very appealing technique -- must it be prohibited under all circumstances? By applying principles B and C cleverly, it is indeed possible to allow the buffering of invalidations. If it is possible to assume that an invalidation is performed, for all practical purposes, as soon as it is latched in the buffer of the destination cache, then the system remains sequentially consistent, provided that the cache gives the invalidation buffer priority over processor requests; in this case invalidations can be considered performed as soon as they are latched in the buffer. This is true since any subsequent accesses by the processor will *always* be serviced after the invalidation.

Above we have shown that principles A, B, and C can be applied to show protocols correct. It is lengthy to reason verbally for every condition of READs and WRITEs, why exactly principles A, B, and C are upheld. Once the principles are comprehended, though, one can easily apply them mentally to a protocol.

## 5. A SYSTEM WITHOUT ATOMIC UPDATES

In this section an architecture and cache coherence protocol are described which do not provide for atomic updates (i.e., instantaneous setting of all read paths by a single broadcast) of data. The architecture is depicted in Figure 4. C clusters of processors are connected via a generic interconnection, which may consist of a crossbar, a hierarchy of buses, a loop, or any type of packet switched network. Within clusters, processors are connected by a single bus. Each cluster contains N processors. Each processor has a private cache that may contain any type of data (including

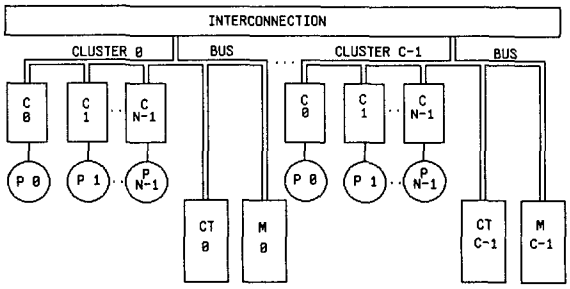



**Figure 4. A cluster-based multiprocessor with private caches.**

shared writable data). Main memory is interleaved on high-order address bits between clusters and on low-order address bits within each cluster. Assume for simplicity and without loss of generality that each memory in a cluster is one single hardware entity. Each memory module contains M words, yielding a total physical memory space of $C \times M$ words. All processors' caches have the same capacity of $K$ blocks, where each block consists of $B$ words. All caches maintain a flag for each block to indicate whether the cache is an owner or a keeper of the block. A *dirty* flag, to indicate that the block has been written to, is not necessary, since all owned blocks are implicitly dirty. A coherence table (CT) is associated with each memory module and is randomly addressed; its size is $\frac{M}{B}$ words. The format of the CT is shown in Figure

5. Each block of a memory module $M_i$ has a corresponding entry in $CT_i$. A CT entry consists of a flag and a field. The flag indicates whether the block is owned or not owned, and the field uniquely identifies the keeper (an owner is also a keeper) where the block may be found. If the keeper is a cache, then the field entry consists of that cache's unique ID, $c$. If several caches are keepers, then the read path may point to any one of them. If the keeper is main memory, then the field's entry is blank (i.e., all 0s). The individual memory module address need not be specified, since it is implied by the block address.

| | | |
|---|---|---|
| BLOCK 0 | K/O | READ PATH |
| BLOCK 1 | K/O | READ PATH |
| BLOCK 2 | K/O | READ PATH |
| BLOCK 3 | K/O | READ PATH |
| | | |
| BLOCK $\frac{M}{B}-1$ | K/O | READ PATH |

**Figure 5. Format of the Coherence Table (CT).**

*5.1 Memory accesses*

In the following, $M_i(x)$ indicates word $x$ of memory module $i$, where memory module $i$ is contained in cluster $i$. Five basic events (read hit, read miss, write hit, write miss, and block replacement) must be accommodated. The protocol uses write-back.

(1) READ HIT: The READ is performed globally as soon as it is processed by the cache controller. This is the case because a cache can never hit on a word whose WRITE has not been globally performed. No external references need to be made.

(2) READ MISS: A cache allocates a block upon a READ miss and becomes a keeper. A READ miss on $M_i(x)$ first blocks the reading processor. A request is sent to $CT_i$. The entry for the block which contains $x$ is checked at $CT_i$, and then the access to that entry is blocked until it is later specifically released. (Any access to a blocked table entry may either be queued or rejected and retried later by the requesting processor, depending on the implementation.) The entry in $CT_i$ specifies which memory device is presently a keeper or an owner.
($a$) If the block is owned by a cache, then the table is modified to show main memory as a keeper, the block at the owner's cache is relabeled as a keeper's copy, and a copy of the block is forwarded to the cache that experienced the original READ miss. The block is also written back to main memory. Upon completion of the transaction, the CT entry is released.
($b$) If the copy pointed at by $CT_i$ is a keeper's copy, then it may simply be copied over from the keeper (main memory) to the new keeper. When the new keeper receives the copy of the block, it sends a signal to the CT to release access to the blocked entry.

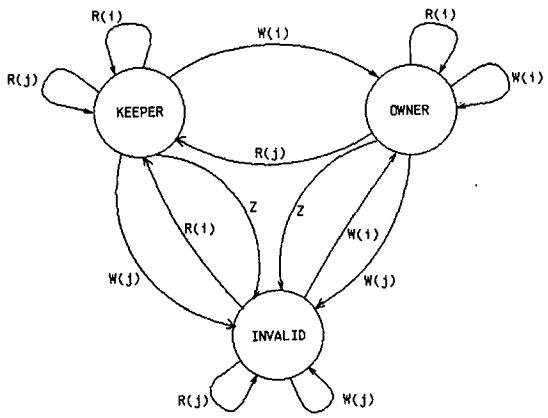
(3) WRITE HIT: Two possibilities exist if a WRITE hit occurs at cache $j$ while accessing word $M_i(x)$: the cache is either a keeper or an owner of the block.
($a$) If the cache is a keeper, it then has to become an owner before the WRITE can be processed at the cache. The $CT_i$ is accessed, the block is locked and the pointer in $CT_i$ is modified to indicate that the block is owned by cache $j$ (i.e., the read path points to $j$). Then invalidations of the block are sent to all clusters, through the cluster interconnection. Cache $j$ changes the status of the block to owner. When all caches have been invalidated (in any sequence), the block entry in $CT_i$ which refers to the block which contains $x$ is made accessible by other requests again. At this point $P_j$ may also proceed to issue another memory access.
($b$) If the cache is an owner, then the WRITE can simply be processed by the cache controller; no other external activity is required. The processor may proceed as soon as the cache has modified the word.

(4) WRITE MISS: A WRITE miss by processor $P_j$ to word $M_i(x)$ first causes processor $P_j$ to block. Then $CT_i$ is accessed for the block information. The read path is followed to the keeper/owner of the block, and a copy of it is forwarded to Cache $j$. The entry is modified to indicate that cache $j$ is now the owner of the block. As usual, the accesses to the modified entry of $CT_i$ are disallowed until a later time. Originally, the block was either owned or kept.
($a$) If the block was owned, previous to the WRITE miss, then when a copy is fetched from that owner, that cache must also be invalidated (this can be done in one operation).
($b$) If the block was kept by several caches before, then not all keepers are explicitly known, and an invalidate must be broadcast to all caches. Once all previous copies of the block are invalidated, requests may be successfully made again to $CT_i$ concerning that block. When cache $j$ receives a copy of the block, it may proceed with the WRITE, and the $P_j$ may continue issuing memory accesses when all other copies of the block have been invalidated.

(5) BLOCK REPLACEMENT: If the cache is a keeper, then the block may be removed without further action. A cache becomes a keeper by either fetching the block from main memory or by fetching it from a previous owner. In either case main memory maintains a valid copy. This is true since when a block is copied from an owner, it is also written back to main memory. If the cache is an owner of the block, then procedure (4a) is followed with the exception that no actual word is updated. Main memory becomes the new owner.
A state transition diagram is shown in Figure 6.

At most times it is indistinguishable whether a read path points at a cache or main memory. Main memory, however, need not keep track of whether the copy of a block that it retains is stale, a keeper's copy, or an owner's copy. This is true since this information is already kept in the appropriate CT. Practical systems will incorporate the table into main memory, such that a table access and a memory access are the same operation.

The reader may verify independently that the conditions for sequential consistency, as defined in sections two and three, are indeed upheld. The reason is that only one update for a particular block may be in progress at any time. The fact that the actual updating of data occurs at different times for different copies is irrelevant, since all




| | | |
|---|---|---|
| R(i) | = | Read by processor i. |
| W(i) | = | Write by processor i. |
| R(j) | = | Read by processor j, i≠j. |
| W(j) | = | Write by processor j, i≠j. |
| Z | = | Block replacement. |

**Figure 6. Transition diagram of data block state in cache i. A cache may be in one of three states w. r. t. a block: (K)eeper, (O)wner, and (I)nvalid.**

accesses may either return an "old" copy of data, or must wait until the "new" data is available to *all* processors. Invalidations must in many cases still be broadcast to all caches. However, a scheme, similar to that in [1], to filter out useless invalidations could be implemented. In a system, though, where the amount of shared, writable data is relatively small or where accesses to such data exhibit good processor locality over time, a very large number of processors may be interconnected efficiently, using a large shared memory space.

*5.2 Indivisible memory accesses*

Sequential consistency guarantees that the order in which accesses are performed with respect to different processors is never reversed. To implement synchronization primitives such as TEST&SET or TEST&TEST&SET, not only must an order between accesses be preserved, but the successive accesses must also be performed *indivisibly*. That is, no other access to the operand may be interleaved between the TEST and SET portions of a TEST&SET access.

Indivisible accesses are easily implemented in bus-based systems, since a processor may hold the bus until its entire atomic access has been performed. While the bus is being held by the processor, no other processor can physically access the memory module or cache in question. Additionally, ownership schemes are easily implemented in bus-based systems -- when such protocols are used, atomicity of accesses is easily implemented. In the system proposed in section 5.1, preventing other processors from making global accesses is not possible, since there exists no unique bus which can be blocked by the processor wishing to perform an indivisible sequence of accesses. However, blocks may be owned and ownership cannot be released without the owner's consent. Switching ownership is also always serialized by the CT. Hence, an atomic access can be correctly executed if the block upon which the access is to be performed is owned by the accessing processor. For all practical purposes, with respect to synchronization accesses, the system reverts to an architecture with a single copy per block architecture. However, the single copy may be located wherever it is most convenient.

The sequence of events which occurs if two processors, $P_i$ and $P_j$, access a binary lock (semaphore) $L$, which may have states of 0 and 1, is described here. Initially $L$=0, which indicates that the lock is open. $P_i$ obtains ownership of the lock by performing a dummy WRITE operation which does not alter the value of the lock. Once the lock is owned by $P_i$ it successfully executes a TEST&SET on it, so that $L$=1. $P_j$ wishes to obtain the lock, it performs a READ on it, with the result that both cache $i$ and cache $j$ become keepers of the lock. $P_j$ then spins on the lock, which is contained in its local cache, and therefore no global activity results. After some time $P_i$ wishes to release the lock. Since this involves a WRITE to $L$, $P_i$ first obtains ownership of the lock and then resets it to 0. On the next TEST of the lock, $P_j$ misses at the cache, so the cache then recopies the unlocked copy, making both caches keepers of the block. $P_j$ finds that the lock is open and proceeds to issue the dummy WRITE, which makes it the owner of the lock. Once $P_j$ is the owner, it may proceed with a TEST&SET operation.

The following is a simple spin_lock algorithm.

```
Spin_lock(x)
  while temp≠0
    {while temp≠0
      temp ←(x)
     temp ←OWN&TEST&SET(x)
    }
```

Note that the "dummy WRITE" which serves to obtain ownership of a block is easily combined with the TEST&SET to form an OWN&TEST&SET operation. The only caveat is that a spinning processor should not use the OWN&TEST&SET to spin on a lock after failing initially in obtaining the lock. Otherwise, several processors spinning on the same lock can cause severe system degradation by passing the lock continuously back and forth. A similar problem may also occur if several locks are "packaged" within the same block.

## 6. CONCLUSION

In this paper a simple condition for sequential consistency of multiprocessor systems was applied to cache coherence protocols. Sequential consistency is a desirable property of general-purpose multiprocessors in which data can be shared; multiprocessors designed to run multitasked programs written in concurrent languages must also adhere to sequential consistency.

To illustrate our approach, we have demonstrated the correctness of several protocols that have been published. These protocols rely on a full broadcast system (either a system bus or an invalidation bus) to propagate updates atomically. Finally, we have described and verified a protocol in which updates are not atomic but propagate from cluster to cluster in a cache-based machine. We have shown that the resulting multiprocessor is not only sequentially consistent but can also execute spin-locks based on indivisible sequences of LOADs and STOREs efficiently.

The contribution of this paper is twofold. First, we have shown how to apply a simple condition to the verification of complex coherence protocols; the condition permits to analyze every step of the protocol. Second, we have demonstrated that *logical* problems associated with large-scale cache-based systems can be solved to provide sequential consistency and reliable execution of indivisible sequences of LOADs and STOREs; these large scale systems are relieved from the condition that *all* processing elements be tied to an invalidation or system bus.

## 7. BIBLIOGRAPHY

[1] L.M. Censier and P. Feautrier,"A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers,* Vol. C-27, No.12, December 1978, pp. 1112-1118.

[2] A.J. Smith, "Cache Memories," *Computing Surveys,* Vol. 14, No. 3, September 1982.

[3] L. Rudolph and Z. Segall, "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors," *11th Int. Symp. on Comp. Arch.,* June 1984, pp. 340-347.

[4] J.R. Goodman, "Using Cache Memory to reduce Processor-Memory Traffic," *Proc. 10th Int. Symp. on Comp. Arch.,* June 1983, Stockholm, Sweden, pp. 124-131.

[5] M. Papamarcos and J. Patel, "A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proceedings of 11th Intl. Symposium on Computer Architecture,* 1984, pp348-354.

[6] J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems,* Vol. 4, No. 4, Nov. 1986

[7] M. Dubois, C. Scheurich and F. Briggs, "Memory Access Buffering In Multiprocessors," *Proceedings of the 13th International Symposium on Computer Architecture,* June 1986.

[8] P. Bitar and A. Despain, "Multiprocessor Cache Synchronization: Issues, Innovations, Evolution," *Proc. of the 13th Ann. Int. Symp. on Comp. Arch.,* June 1986.

[9] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing,* Mc Graw-Hill, 1984.

[10] S.J. Frank, "Synapse Tightly Coupled Multiprocessors -- A New Approach that Solves Old Problems," *Proceedings of NCC,* Las Vegas, 1984.

[11] M. Hill et al., "Design Decisions in Spur," *IEEE Computer,* November 1986.

[12] G.R. Andrews, *et al.,* "Concepts and Notations for Concurrent Programming," *Computing Surveys,* Vol.15, No. 1, March 1983.

[13] H.T. Kung, "Synchronized and Asynchronous Parallel Algorithms for multiprocessors," in *Algorithms and Complexity: New Directions and Recent Results,* J.F. Traub Ed., New York: Academic Press, 1976.

[14] G.M. Baudet, "The Design and Analysis of Algorithms for Asynchronous Multiprocessors," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, August 1978.

[15] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communication of the ACM,* July 1978, Vol. 21, No. 7.

[16] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers,* Vol. C-28, No. 9, September 1979, pp. 690-691.

[17] W. W. Collier, "Reasoning about Parallel Architectures," submitted to *JACM,* 1985.

[18] P. M. Vitanzi and B. Auerbach, "Atomic Shared Register Access by Asynchronous Hardware," Conference on the Foundations of Computer Sciences, 1986.

[19] M. Dubois and C. Scheurich, "Dependency and Hazard Resolution in Multiprocessors," *submitted to IEEE Transactions on Software Engineering.* Also, Univ. of Southern Cal. Technical Report CRI 86-20.

[20] M. Dubois and F.A. Briggs, "Effects of Cache Coherency in Multiprocessors," *IEEE Transactions on Computers,* Vol. C-31, No.11, November 1982.